COS 597A:
Principles of
Database and Information Systems

Indexing files

Last time

- File = a collection of (pages of) records
- File organizations: two issues
  - how records assigned pages
  - how pages put on disk
  - Heap: linked list (or directory) of pages
    - records anywhere on any page - no order
    - pages anywhere on disk
  - Sorted sequential pages
    - Records sequential in each page by designated sort attribute
    - can binary search: get i^{th} page in one disk read
  - Hashing:
    - Designated hash attribute hashing records to buckets
    - Bucket => (primary) page for hash function value
      - pages can be anywhere if hash table gives location

Focus on key elements of cost

Improvements only for attribute of sort or hash
Improve access using other attributes? => index

Avg. time	Heap	Sorted	Hashed
Search = (unique)	.5BD	$D\log_2 B$	D
Search range	BD	**$D(\log_2 B$ + # extra matching pages)**	1.25 BD
Insert	2D	**Search + D + BD**	2D
Delete (have record location)	2D	2D+BD	2D

B data pages in file **D** avg time to R/W page **R** records per page

Index

- Auxillary information on location of a record or page to facilitate retrieval
- Search key: attribute (i.e. field, column) used as look-up value for index
  - not confuse with {primary, candidate, super} key
  - alternate term "index field"
    - "index key" if attribute is a candidate key
  - Could actually be combination of attributes
    - e.g. LastName, FirstName
- Basic index is a file containing mappings:
  Seach key value → pointer(s) to page(s) containing records with given search key value

Index Types

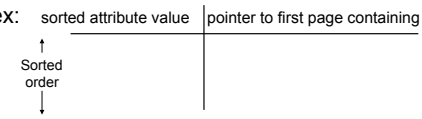
1. Index works *with* file organization
   - Index and file work off same attribute
   - Two types:
     A. Index **is** file organization
        - Example: Hashing file organization
        - Index is access method: get pointer to page serving as primary bucket of records for given hash value
     B. Index supplements file organization
        - Example: Sequential file plus search tree whose leaves point to first page containing value seeking
   - called clustered index
   - some refer to as primary index
     - not necessarily on primary key of relation

Index Types cont.

2. Index works independent of file organization
   - File not organized on search key of index
   - Index must provide
     search key value → list of pointers to *all* file pages that contain records with that value
   - Example hash index:
     - bucket contains list of page pointers
     - pages may be scattered throughout the file
     - overflow if too many pointers for one bucket
   - called nonclustering index
   - some refer to as secondary index

A Sorted Index

- Consider sorted file but without consecutive pages stored adjacently on disk
  - Each page sorted
  - Each page linked to next page in sorted order
  - ***Cannot*** binary search
- Index:



- One entry per attribute value in data file => dense index
- Can binary search index entries if can keep in memory or in sequential disk pages

7

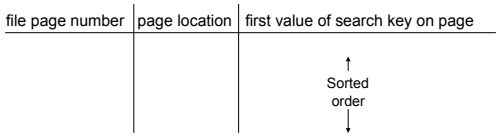
Indexing sorted files - notes

- When index on sorted file using same attribute, index need not be dense (so sparse)
- Insert/delete for sorted file with sorted index costs to maintain sorted order in both
- Index may be sorted on different attributes(s) than file, but clustered as file is
  - Example: file sorted on (last_name, first_name)
    index sorted on last_name

8

Alternative sparse index for sorted file

again:
index search key same as sort attribute for file



One entry *per file page*
Again, binary search if keep in memory or sequentially on disk

9

Compare costs:

dense sorted index **versus**
sparse sorted index with one value per data file page

- Use our crude estimates with
  B data pages in file **D** avg time to R/W page
  R records per page
- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key
- Cost search for unique value using dense index?

- Cost search for unique value using sparse index?

10

Cost example dense sorted index

- Use our crude estimates with
  B data pages in file **D** avg time to R/W page
  R records per page
- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key

- Cost search for unique value using dense index:

  number of records is the same for index file
  B/10 pages in index file (file page size is fixed for all files)
  Binary search cost = $D\log_2(B/10)$

  Total cost = $D\log_2(B/10) + D$
  includes data page access

11

Cost example sparse sorted index

- Use our crude estimates with
  B data pages in file **D** avg time to R/W page
  R records per page
- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key

- Cost search for unique value using sparse index:

  B pages in data file => B entries in index file
  10R index records per file page => B/(10R) index pages
  Binary search cost = $D\log_2(B/(10R))$

  Total cost = $D\log_2(B/(10R)) + D$
  includes data page access

12

Compare costs:

- Use our crude estimates with
  B data pages in file **D** avg time to R/W page
  R records per page
- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key
- Cost search for unique value using dense index?

$$D\log_2(B/10) + D$$

- Cost search for unique value using sparse index?

$$D\log_2(B/(10R)) + D$$

13

Compare costs: insertion

- Use our crude estimates with
  B data pages in file **D** avg time to R/W page
  R records per page
- Suppose index record 1/10 size of data record
- Suppose search key (= sort attribute) is candidate key
- Recall data file pages not nec. stored consecutively on disk
  - so can use overflow pages
- Cost to insert = cost to insert in data file
  + cost to insert in index file

= Search cost
+ D + ~4D write data file page and move ~1/2 records of page if overflow
+ D write index entry
+ { D*B/10 move records for dense index
  { D*B/(10R) move records for sparse index

14

Index independent of file organization

But look again,
if search key is a *candidate key*,
this index works for *any* file organization :

search key	pointer to *unique* page containing
↑ Sorted order ↓	

One entry per search key value - dense
Can binary search index as before if keep in memory or sequentially on disk

15

Sorted index for general case

- One value of search key found in many records
- Need list of pointers to pages containing these records
- Dense index still works
- Most common arrangement:
  - indirection

Seach key	pointer to page containing list
↑ One entry per attribute value. Sorted order ↓	

16

Addressing costs

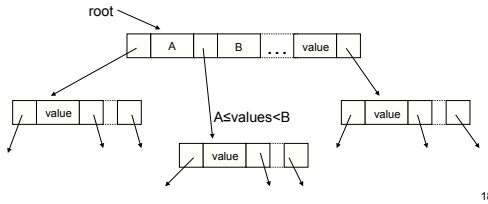
- Large sorted index costly in space and in time to insert/delete
  - When sorted index clustered, can use sparse index to avoid space
  - For general case, *must* have dense index
- Ideal: index to fit on one file page.
  - Keep in main memory
- Rarely achieve, so next best:
  - Index need not be stored sequentially on disk
  - Access cost is no worse than $O(\log_2 B)$

  => **Search Tree!**

17

Tree index

- Each node of tree fits in one page
- Each node of tree contains search key values and pointers to subtrees for ranges of values
- A leaf is
  - For clustered index: a page of data file
  - For general index: a page of pointers to records with given index values



18

Static Trees

- Build for file of records as balanced tree
- Not gracefully accommodate insert/delete
- ISAM: Indexed Sequential Access Method

- We focus on dynamic search trees

19

Dynamic Trees

- Tree will change to keep balance as file grows/ shrinks
- Tree height: longest path root to leaf
- N data entries
  - Data entry is page of data file if clustered index
  - Data entry is page of (value, record pointer) pairs otherwise
- Want tree height proportional to logN always

20

B+ Trees

- Most widely used dynamic tree as index
- Most widely used index

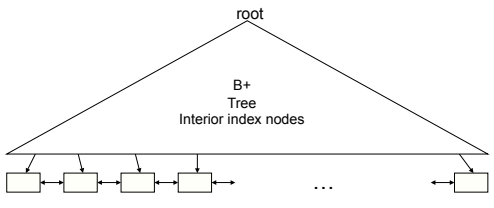
- Properties
  - Data entries only in leaves
    - Compare B-trees
  - One page per tree node, including leaves
  - All leaves same distance from root => balanced
  - Leaves doubly linked
    - Gives sorted data entries
  - Call search key of tree "B+ key"

21

B+ trees continued

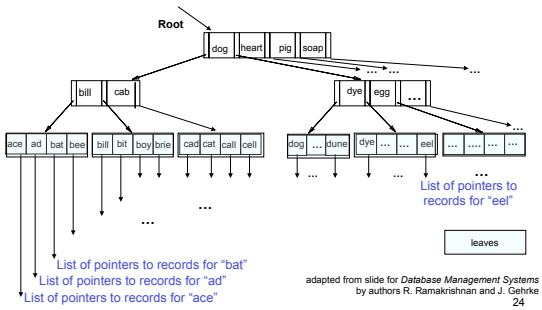
- To achieve equal distance all leaves to root cannot have fixed fanout
- To keep height low, need fanout high
  - Want interior nodes full
- Parameter d - order of the B+ tree
- Each interior node except root has m keys for d≤m≤2d
  - m+1 children
- The root has m keys for 1≤m≤2d
  - Tree height grows/shrinks by adding/removing root
- d chosen so each interior node fits in one page

22



Leaves will be 1/2 full to full as well

23



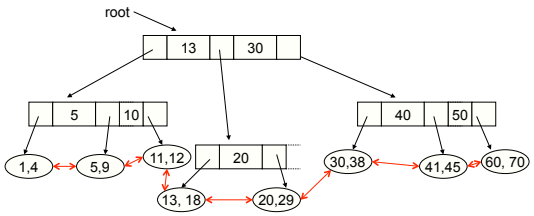
adapted from slide for *Database Management Systems* by authors R. Ramakrishnan and J. Gehrke

24

B+-tree insert and delete
Example

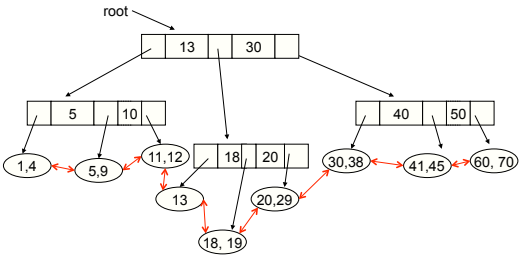
25

Starting configuration
B+ tree of order d=1



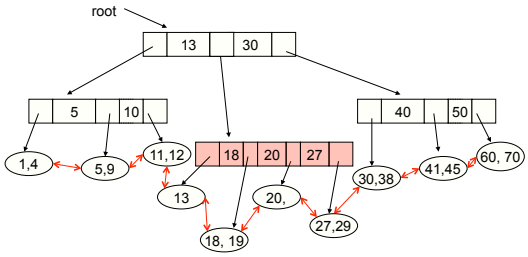
26

Insert 19:
split leaf; expand parent with key 18



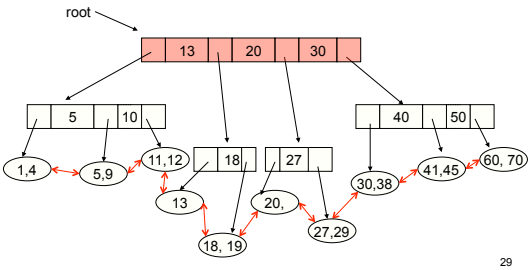
27

Insert 27
split leaf; expand parent with key 27 => too full



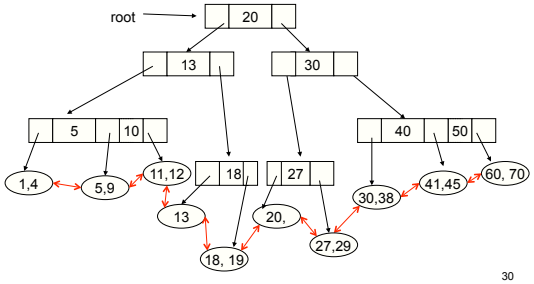
28

Insert 27
split leaf; split parent;
expand grandparent with key 20 => too full



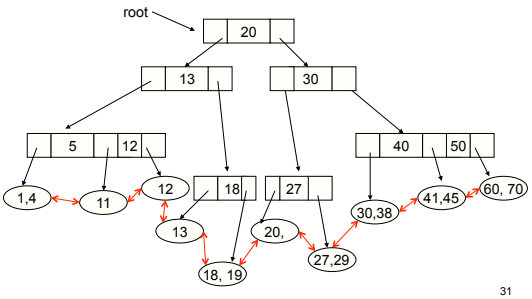
29

Insert 27
split leaf; split parent; split grandparent
new root with key 20



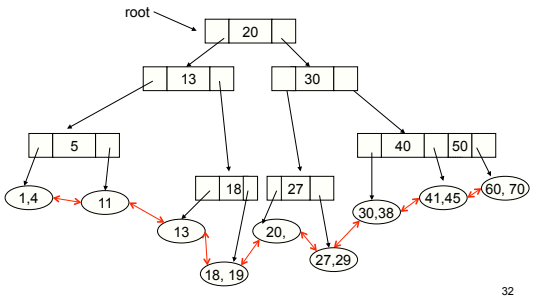
30

Delete 5, then 9
redistribute from right sibling



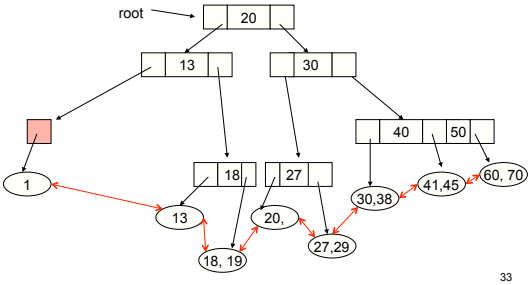
31

Delete 12
merge leaves, delete key from parent



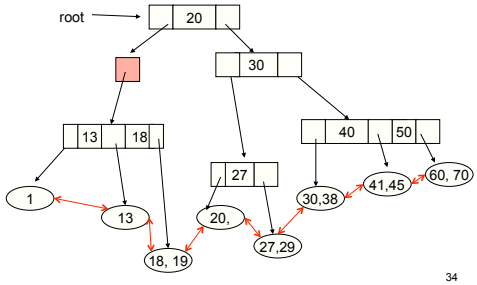
32

Delete 4, then 11
merge leaves, delete key from parent
=>parent not full enough



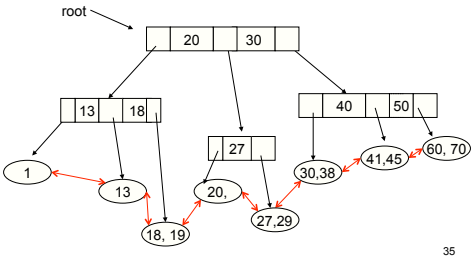
33

Delete 4, then 11
merge leaves, merge parent, bringing down key 13
=>grandparent not full enough



34

Delete 4, then 11
merge leaves; merge parent, bringing down key 13
merge grandparent, bring down key 20,
remove root



35